A Highly Scalable Parallel Boundary Element Method for Capacitance Extraction

Yu-Chung Hsiao *
Massachusetts Institute of Technology
yuchsiao@mit.edu

Luca Daniel
Massachusetts Institute of Technology
luca@mit.edu

ABSTRACT

Traditional parallel boundary element methods suffer from low parallel efficiency and poor scalability due to the long system solving time bottleneck. In this paper, we demonstrate how to avoid such a bottleneck by using an instantiable basis function approach. In our demonstrated examples, we achieve 90% parallel efficiency and scalability both in shared memory and distributed memory parallel systems.

Categories and Subject Descriptors

J.6 [**Computer-aided engineering**]: Computer-aided design (CAD)

General Terms

Algorithms.

Keywords

Parallel computing, Boundary element method, Capacitance extraction, Field solver.

1. INTRODUCTION

Standard boundary element methods usually involve two steps: the system setup step, in which a linear system is constructed, and the system solving step, respectively. The time complexity for these two steps are $O(N^2)$ and $O(N^3)$ if direct methods are used. Such complexity is determined by the boundary element method formulation, irrespective of the choice of basis functions. Among various types of basis functions, the most prevalent choice is piecewise constant basis functions, because of the availability of closed-form expressions for both collocation and Galerkin's integrations. Piecewise constant basis functions are also widely applicable for representing any general physical charge distributions. However, using such type of basis functions inevitably results in a large linear system, which requires extremely long system solving time.

In order to solve large linear systems efficiently in single-core computation environments, several acceleration methods based on piecewise constant basis functions were proposed during the last couple of decades, such as fast multipole expansion [4] and pre-corrected FFT [6]. Instead of using direct Gaussian elimination, these methods use Krylov subspace iterative methods, and exploit specialized algorithms to approximate matrix-vector products, reducing both time and memory complexity from $O(N^2)$ to $O(N \log N)$.

Such acceleration algorithms are, however, not efficiently parallelizable. An efficient parallel algorithm requires an "embarrassingly parallelizable" structure, which means low data dependency, minimized memory accesses, and small amount of data transfer. In Krylov subspace iterative methods, due to the use of piecewise constant basis functions, the residual vectors are large and need to be transfered between parallel computing nodes or accessed by different threads in shared memory for each iteration. Besides, although standard matrix-vector product operations are embarrassingly parallelizable, their approximated forms adopted in the aforementioned acceleration algorithms introduce large data dependency when reusing partial calculations in order to achieve acceleration. Some previous works, such as [1] and [7], showed that the parallel performance of such matrix-vector approximations is quite limited: the parallel speedup saturates very quickly, and the efficiency drops to 60% or even less at around only eight parallel computing nodes.

All these observations give rise to the need for using a more compact solution representation than piecewise constant basis functions. When a much smaller number of basis functions, N, is used to represent the solution, direct solving methods are considered efficient. This is because direct methods do not suffer from initialization overheads from which the matrix-vector approximation methods usually suffer. In addition, real solving performance heavily depends on whether the memory hierarchy is utilized efficiently, for instance, by considering the machine-specific cache size in order to reduce the cache miss rate. In practice, direct solving methods can be computed much more efficiently if the underlying basic linear algebra operations are optimized with the consideration of hardware design. Various mature linear algebra libraries, for instance, ATLAS, GotoBLAS, and other hardware vendor optimized routines, provide more than one order of magnitude faster performance than the implementation without hardware consideration. The aforementioned matrix-vector approximation

*This work is supported by Mentor Graphics, Advanced Micro Devices (AMD), the Global Research Collaboration (GRC), and the Interconnect Focus Center (IFC).

DAC 2011, June 5-10, 2011, San Diego, California, USA.

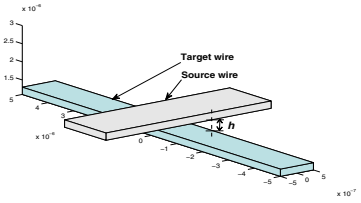

Figure 1: A pair of simple crossing wires.

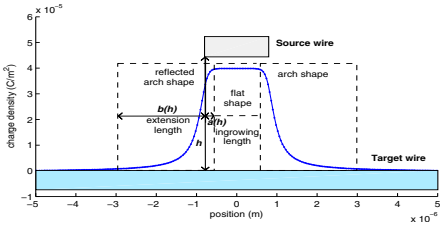

Figure 2: Extracted flat and arch shapes.

methods, which utilize non-standard matrix operations, are not compatible with and do not benefit from such highly optimized linear algebra libraries.

However, using a compact solution representation is not always advantageous. Its compactness usually results from using complex basis function shapes, which complicates the numerical integrations in the system setup step. Fortunately, when N is small, the linear system can be stored in the matrix form, and the system setup step consists of simply filling in the matrix. Each entry of such matrix can be computed independently in each parallel computing node without data dependency and communication overheads. Therefore, highly efficient and scalable parallelization can be achieved.

In this paper, we used instantiable basis functions [3] as the compact solution representation for the capacitance extraction problem. Using such basis functions in a single-core environment would result in negligible system solving time but relatively long system setup time, which is an excellent scenario for efficient parallel computing.

The background of boundary element methods for the capacitance extraction problem and instantiable basis functions are summarized in Section 2. Section 3 discusses a parallel matrix filling strategy that minimizes data dependency and memory accesses. In Section 4, we developed four generic acceleration techniques for computing integrations, whose importance is usually overlooked when piecewise constant basis functions are used. When more complex basis functions are used and the system setup time dominates, the acceleration of computing integrations can directly improve the overall performance. Section 5 describes our C++ implmentation in share-memory and distributed-memory systems through OpenMP and MPI, respectively. In Section 6, their performance is presented. Our conclusions are made in Section 7.

2. BACKGROUND

2.1 The BEM for capacitance extraction

Given an n-conductor interconnect geometry embedded in a uniform dielectric medium, the capacitance matrix can be obtained by solving the integral equation of the electric static problem

$$\int_{S'} \frac{\rho(\mathbf{r}')}{4\pi\varepsilon\|\mathbf{r}-\mathbf{r}'\|}\mathrm{d}s' = \phi(\mathbf{r}) \qquad (1)$$

for charge distribution $\rho(\mathbf{r}')$. In equation (1) , $\mathbf{r}$ and $\mathbf{r}' \in \mathbb{R}^3$ are position vectors in 3D space, $\phi(\mathbf{r}) : \mathbb{R}^3 \mapsto \mathbb{R}$ is the electric potential, $\rho(\mathbf{r}') : \mathbb{R}^3 \mapsto \mathbb{R}$ is the unknown charge distribution, S' is the conductor surfaces, ε is the dielectric constant of the medium, and the operator $\|\cdot\|$ computes the Euclidean distance of its argument. By expressing $\rho(\mathbf{r}')$ as a linear combination of basis functions ψ_j for $j = 1, 2, \ldots, N$, and applying the standard Galerkin's testing, one can transform equation (1) into

$$\sum_{j=1}^{N}\left[\int_{s_i}\int_{s'_j}\frac{\psi_i(\mathbf{r})\psi_j(\mathbf{r}')}{4\pi\varepsilon\|\mathbf{r}-\mathbf{r}'\|}\mathrm{d}s'\mathrm{d}s\right]\rho_j = \int_{s_i}\psi_i(\mathbf{r})\phi(\mathbf{r})\mathrm{d}s, \qquad (2)$$

where s_i and s'_j are the supports of ψ_i and ψ_j, respectively. Equation (2) can be further represented as a system of linear equations

$$P\rho = \Phi, \qquad (3)$$

where $\rho = [\rho_1, \rho_2, \ldots, \rho_N]^T$, P is an $N \times N$ system matrix consisting of the bracketed factors for all i, j in equation (2), and Φ is an $N \times n$ matrix. The k-th column of Φ is the right-hand side of equation (2) with

$$\phi(\mathbf{r}) = \begin{cases} 1 & \text{, if } \psi_i \text{ is on the } k\text{-th conductor} \\ 0 & \text{, otherwise.} \end{cases}$$

After equation (3) is solved, the capacitance matrix C can be found by computing the expression $C = \Phi^T\rho$.

2.2 Instantiable basis functions

Unlike piecewise constant or other mathematically defined basis functions, instantiable basis functions [3], are the collection of the fundamental shapes extracted from elementary problems, such as a pair of crossing wires shown in Figure 1. The curve shown in Figure 2 is the side view of the charge distribution on the top face of the bottom wire induced by the top wire. This charge distribution can be further decomposed into a constant flat shape and two arch shapes $A_{\mathbf{p}}(u)$. The subscript $\mathbf{p}$ is a vector of parameters. The vector $\mathbf{p}$ contains wire separation h in Figure 1 and other geometric parameters, depending on the required capacitance accuracy. These shapes are then extended into 2D flat and arch templates by adding a perpendicular coordinate with constant values, i.e., $T_F(u, v) = 1$ and $T_{A_{\mathbf{p}}}(u, v) = A_{\mathbf{p}}(u)$, where v is in the direction into the paper.

The full set of instantiable basis functions consists of two parts: induced basis functions and face basis functions. Face basis functions are by default placed on each rectangular surface of a conductor and have a normalized value of 1. Induced basis functions are used to capture the electrostatic interactions between wires. They are instantiated by assembling flat and arch templates with proper parameter vectors $\mathbf{p}$ and placed in the neighborhood of wire intersections. Instantiable basis functions are constructed under the assumption that all conductors are embedded in a uniform dielectric medium and in Manhattan geometry.

3. PARALLELIZATION

By using instantiable basis functions, more than 95% of the total runtime is spent on the system setup step. In the following, we will introduce a balanced workload division scheme with minimized data communications for constructing the system matrix P. For the system solving step, we will resort to the standard direct method implemented in multithreaded linear algebra libraries.

A single instantiable basis function consists of one or multiple templates,

$$\psi_{i'}(\mathbf{r}) = \sum_{\bar{i}} \psi_{i',\bar{i}}(\mathbf{r}).$$

Therefore, for N instantiable basis functions constructed from a given capacitance extraction problem, a matrix entry of $P \in \mathbb{R}^{N\times N}$ can be expressed as the summation of the integrals involving every template pair:

$$P_{i',j'} = \sum_{\bar{i}}\sum_{\bar{j}} \int_{s_{i',\bar{i}}} \int_{s_{j',\bar{j}}} \frac{\psi_{i',\bar{i}}(\mathbf{r})\psi_{j',\bar{j}}(\mathbf{r})}{4\pi\epsilon\|\mathbf{r}-\mathbf{r}'\|} \mathrm{d}s'\mathrm{d}s. \quad (4)$$

Each $\psi_{k',\bar{k}}$ is either an arch template $T_{A_\mathbf{p}}(u,v)$ or a constant value of 1 if it is a face basis function or a flat template. When both $\psi_{i',\bar{i}}$ and $\psi_{j',\bar{j}}$ are constant, which is the most common situation, the corresponding integral degenerates to the Galerkin's integration of piecewise constant basis functions. The computation becomes much more balanced if each integral is computed based on templates rather than basis functions. Accordingly, we collect all templates from each basis function and relabel them from 1 to M. A matrix $\tilde{P} \in \mathbb{R}^{M\times M}$ can be constructed such that

$$\tilde{P}_{ij} = \int_{s_i}\int_{s_j} \frac{T_i(\mathbf{r})T_j(\mathbf{r})}{4\pi\varepsilon\|\mathbf{r}-\mathbf{r}'\|}\mathrm{d}s'\mathrm{d}s, \quad (5)$$

and then condensed into P by combining the rows and the columns which are related to the same basis function. Figure 3 shows an example that a matrix P is constructed by four basis functions ($N = 4$) in which only ψ_3 consists of two templates ($M = 5$). The mapping between $\psi_{i',\bar{i}}$ and T_i is defined as in the following order set:

$$\{\psi_{1,1}, \psi_{2,1}, \psi_{3,1}, \psi_{3,2}, \psi_{4,1}\} = \{T_1, T_2, T_3, T_4, T_5\}.$$

In practice, M is usually 1.2 to 3 times greater than N, or equivalently, $\tilde{P}$ can be 9 times larger than P. In order to minimize memory access, we avoid allocating $\tilde{P}$ but construct P directly. Figure 3 shows the numbers of entries of $\tilde{P}$ that are summed to a single entry of P by different color levels. Because $\tilde{P}$ is symmetric, only those off-diagonal entries of $\tilde{P}$ which are combined to the diagonal of P contribute their values twice, for instance $\tilde{P}_{i=3,j=4}$ in Figure 3. This statement is generally true irrespective of the number of templates included in a basis function.

In order to achieve high parallel efficiency, it is sufficient to construct system matrix P through equation (5) and divide work according to the number of entries in $\tilde{P}$. Although the cost of calculating each $\tilde{P}_{ij}$ is influenced by template types and spatial orientations, in practice, such work division is sufficiently balanced as we will see in Section 6.

The resulted parallelization algorithm is constructed by using an independent index k which iterates the upper triangular matrix of $\tilde{P}$ from 0 to $M(M+1)/2-1$. The index k is converted into (i,j) for $\tilde{P}$ and then mapped to (i',j') for P. For a parallel system of D computing nodes, the full range of k is equally divided into D partitions, and each node is in charge of calculating the corresponding entries of $\tilde{P}$ in each partition. This parallelized system setup algorithm is summarized in Algorithm 1.

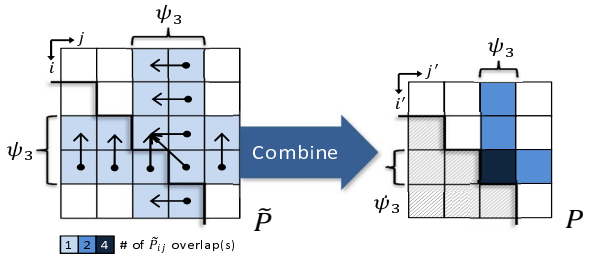

Figure 3: Matrix $\tilde{P}$ condenses into P.

Algorithm 1 Parallelization prototype for system setup

Require: D parallel nodes indexed from $d = 1$ to D;
  Allocate and initialize $P = 0$;
  Construct an array l such that $l_i = i'$;
  $K \longleftarrow M(M+1)/2$;
  Partition $\{0, 1, \ldots, K-1\}$ into $K_0, K_1, \ldots, K_{D-1}$ s.t.
    $|K_1| = |K_2| = \cdots = |K_{D-1}| = \lfloor K/D \rfloor$ and
    $|K_D| = K - (D-1)|K_1|$;
  for each parallel node $d \in \{1, \ldots, D\}$ **do**
    for all $k \in K_d$ **do**
      $j \longleftarrow \lfloor(-1+\sqrt{1+8k})/2\rfloor$;
      $i \longleftarrow k - j'(j'+1)/2$;
      $\tilde{p} \longleftarrow$ Galerkin's integration of T_i and T_j;
      if $i \neq j$ and $l_i = l_j$ **then**
        $P_{l_i,l_j} \longleftarrow P_{l_i,l_j} + 2\tilde{p}$;
      else
        $P_{l_i,l_j} \longleftarrow P_{l_i,l_j} + \tilde{p}$;
      end if
    end for
  end for
  return P;

4. INTEGRATION ACCELERATION TECHNIQUES

After the system setup work is dispatched to each parallel computing node, the integral in the square bracket of equation (2) is computed in a sequential environment. Although we mainly address the electrostatic problem in this work, the techniques we develop in this section are also applicable to other boundary element methods, as long as their corresponding Green's functions decay with distance. In the following, we will describe the acceleration techniques for computing the integral

$$\tilde{P}_{i,j} = \int_{y_{i1}}^{y_{i2}}\int_{x_{i1}}^{x_{i2}}\int_{y'_{j1}}^{y'_{j2}}\int_{x'_{j1}}^{x'_{j2}} \frac{T_i(x,y)T_j(x',y')}{4\pi\varepsilon\sqrt{\delta x^2+\delta y^2+\delta z^2}}\mathrm{d}s'\mathrm{d}s, \quad (6)$$

in which $\delta u = u - u'$ for $u \in \{x, y, z\}$, and $\mathrm{d}s$ and $\mathrm{d}s'$ are $\mathrm{d}x\mathrm{d}y$ and $\mathrm{d}x'\mathrm{d}y'$, respectively. The support of $T_i(x,y)$ is $[x_{i1}, x_{i2}] \times [y_{i1}, y_{i2}]$. The support of $T_j(x',y')$ is defined similarly.

4.1 General considerations

In instantiable basis functions, T_i and T_j are defined to have at most 1D shape variation. When $T_i(x,y) = T_i(x)$

and $T_j(x', y') = T_j(y')$, we can rearrange the expression in equation (6) into

$$\int_{x_{i1}}^{x_{i2}} T_i(x) \int_{y'_{j1}}^{y'_{j2}} T_j(y') \left[\int_{y_{i1}}^{y_{i2}} \int_{x'_{j1}}^{x'_{j2}} \frac{\mathrm{d}x'\mathrm{d}y/(4\pi\varepsilon)}{\sqrt{\delta x^2 + \delta y^2 + \delta z^2}} \right] \mathrm{d}y'\mathrm{d}x. \tag{7}$$

The inner bracketed 2D integration in equation (7) is identical to the collocation integration with piecewise constant basis functions, which has a closed-form expression and can be computed efficiently. The remaining outer 2D integration is then computed numerically by Gaussian quadrature. The analytical expressions are also available for 3D or 4D when one or both of T_i and T_j are flat templates. However, the computation cost grows rapidly as the integration dimension increases. For instance, a 2D expression involves only 8 terms whereas more than 100 terms are present in a 4D expression. In terms of runtime, the former is roughly 10 times faster than the latter. A useful strategy to lower the chance of invoking high dimensional expressions is to utilize the decay property of the Green's function with distance. When two templates are separated far enough, the function behaviors of the higher and the lower dimensional expressions are numerically indistinguishable. Therefore, we can use the lower dimensional expression to approximate the higher dimensional integral. The distance above which we can start to perform such approximation (approximation distance in short) can be parameterized as a function of template parameters. Additional approximation levels can also be inserted between integration dimensions by using quadrature points to further reduce the chance of invoking higher dimensional expressions.

4.2 Acceleration techniques

The integral in (7) can be decomposed into two parts: the outer Gaussian quadrature and the inner closed-form expressions. Evaluating such closed-form expressions is the bottleneck of overall performance. In this section, we propose four different acceleration techniques that can provide an extra speedup in addition to parallelization.

4.2.1 Direct tabulation

Instead of evaluating 100 terms in the 4D analytical expression, we can tabulate the expression as a 6-parameter table. Such tabulation is feasible because the range for each parameter we need to tabulate is limited by the approximation distance. Beyond the approximation distance, we can use the lower dimensional expressions which can also be tabulated. The target tabulation function is

$$F_{\text{definite}} = \int_{y_1}^{y_2} \int_{x_1}^{x_2} \int_{y'_1}^{y'_2} \int_{x'_1}^{x'_2} \frac{1}{4\pi\varepsilon\|\mathbf{r} - \mathbf{r}'\|} \mathrm{d}x'\mathrm{d}y'\mathrm{d}x\mathrm{d}y. \tag{8}$$

One of the attractive features of this method is its very manageable error control. However, its performance is limited by the expensive six-dimensional linear interpolation.

4.2.2 Tabulation of indefinite integrals

We can reduce the tabulation parameters from six to three by tabulating the indefinite integral without substituting the upper and lower limits of each dimension. Equation (8) can be rewritten as an indefinite integral form:

$$F_{\text{indefinite}}(x - x', y - y', z)\Big|_{x'_1}^{x'_2}\Big|_{y'_1}^{y'_2}\Big|_{x_1}^{x_2}\Big|_{y_1}^{y_2} = F_{\text{definite}}. \tag{9}$$

The error control of this formula needs careful investigation of function behavior.

4.2.3 Tabulation of expensive subroutines

When the closed-form expressions are evaluated, most of the computation time is spent on calling expensive elementary functions, such as `atan` and `log`. Tabulating these single parameter functions is memory efficient even with high accuracy requirement and using zero-order hold. The IEEE-754 floating-point representation is especially useful for tabulating the logarithmic function

$$\log_2(\{\text{mantissa}\} \times 2^{\{\text{exp.}\}}) = \{\text{exp.}\} + \log_2(\{\text{mantissa}\}),$$

in which only $\log_2(\{\text{mantissa}\})$ needs to be tabulated [5]. In our experiment, tabulating the first 14 bits of mantissa is sufficient to achieve an error below 1% for the 4D analytical expression. This approach is 5 times faster than the built-in `log` function on a Xeon 3.2 GHz machine. A similar tabulation for `atan` provides a 4 times speedup compared to its built-in version. This method is attractive for its easy implementation and error control.

4.2.4 Rational fitting

It can be seen from equation (9) that the analytical expression of an integral is ill-conditioned. Several most significant digits of the indefinite integrals are canceled out when subtracting the lower limit substitution from the upper limit substitution. Instead, we can adopt a multivariable rational function of degree (n, m) as a more efficient and numerically stable expression

$$f(\mathbf{w}) = \frac{f_N(\mathbf{w})}{f_D(\mathbf{w})}, \tag{10}$$

where $\mathbf{w} \in \mathbb{R}^k$ is an input vector, and $f_N(\mathbf{w}), f_D(\mathbf{w}) \in \mathbb{R}^k \mapsto \mathbb{R}$ are polynomial functions defined as

$$f_N(\mathbf{w}) = \sum_{|\alpha| \le n, \forall \alpha} \beta_{N,\alpha} \mathbf{w}^{\alpha}, \tag{11}$$

and

$$f_D(\mathbf{w}) = \sum_{|\alpha'| \le m, \forall \alpha'} \beta_{D,\alpha'} \mathbf{w}^{\alpha'},$$

in which α and α' are k-dimensional multi-indices. Using such rational function form is especially suitable for those Green's functions which decay with distance. To find the best coefficients β_N and β_D, it is sufficient to reformulate equation (10) into an optimization problem and apply n training samples $\tilde{f}(\mathbf{w}_i), i = 1, 2, \ldots, n$:

$$\begin{aligned} \underset{\beta_{N,\alpha}, \beta_{D,\alpha'}}{\text{minimize}} \quad & \sum_{i=1}^{n} |\tilde{f}(\mathbf{w}_i) f_D(\mathbf{w}_i) - f_N(\mathbf{w}_i)| \\ \text{subject to} \quad & \sum_{|\alpha'| \le m, \forall \alpha'} \beta_{D,\alpha'} = 1. \end{aligned} \tag{12}$$

This optimization problem can be solved by using STINS [2]. The constraint in equation (12) reduces the degrees of freedom of $\beta_{N,\alpha}$ and $\beta_{D,\alpha'}$ by 1 in order to normalize the rational function. The error control of this approach relies on the choice of training samples.

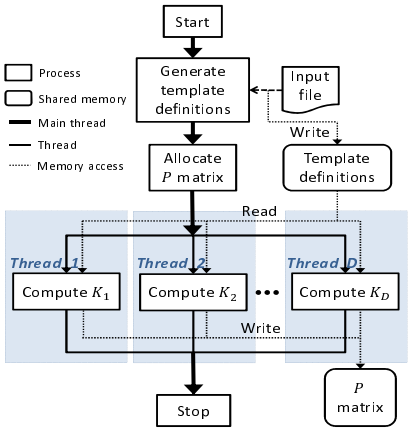

Figure 4: Flowchart for the system setup step in a shared-memory system.

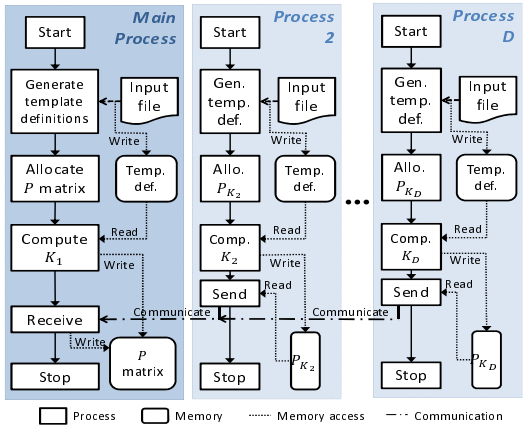

Figure 6: Flowchart for the system setup step in a distributed-memory system.

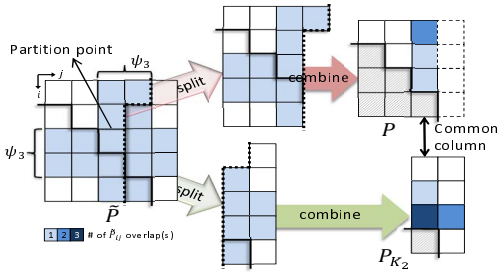

Figure 5: Partitions of system matrix P in a two-node distributed-memory system.

4.3 Performance comparison and discussion

To compare the performance of the four integration techniques in Section 4.2, we focus on a simplified 2D analytical expression

$$f_{2D}(\mathbf{r}) = \int_{y_1'}^{y_2'} \int_{x_1'}^{x_2'} \frac{1}{\sqrt{\mathbf{r}-\mathbf{r}'}} \mathrm{d}x'\mathrm{d}y'. \qquad (13)$$

The performance for each method is listed in Table 1. All these results are tested on a Xeon 3.2 GHz machine implemented in C++ single precision floating points with 1% error tolerance.

Table 1: Performance comparison of different integration acceleration techniques for equation (13)

Techniques	Time/speedup	Memory
0. Original analytical expr.	280 ns/1.00×	≈ 0
1. Direct tabulation	136 ns/2.06×	1.5 MB
2. Tabulation of indef. int.	240 ns/1.16×	2.3 MB
3. Tabulation of exp. routines	128 ns/2.20×	2.0 MB
4. Rational fitting	224 ns/1.24×	≈ 0

It should be noted that this performance assessment is only valid for instantiable basis functions. This is because instantiable basis functions allow templates to overlap each other, which is generally not the case for other types of basis functions. As a result, the minimum degree of rational functions and the size of the indefinite integral table are greatly increased to account for the case when equation (13) is evaluated near the boundary of the integral region $[x_1', x_2'] \times [y_1', y_2']$. We chose the method of tabulating expensive subroutines in our implementation.

5. IMPLEMENTATION

We implemented Algorithm 1 in both shared-memory and distributed-memory systems through OpenMP and MPI, respectively.

5.1 In a shared-memory system

In this case, both template definitions from the input file and the output system matrix P are stored in shared memory. $D-1$ additional threads are generated after P is allocated. The thread d retrieves template definitions from shared memory, computes the entries of $\tilde{P}$ in the partition K_d within its private memory, and then adds the result to the corresponding entries of P. Multithreads are restored into a single main thread after every K_i is computed. This process is diagrammed in Figure 4.

5.2 In a distributed-memory system

In a distributed-memory system of D nodes, the main process $d = 1$ computes the entries of $\tilde{P}$ in the partition K_1 and stores the results in P as in a sequential environment. For $d \neq 1$, the process d holds its own copy of template definitions, calculates entries of $\tilde{P}$ in K_d individually, stores the results in a separate partial matrix P_{K_d} of P, and finally send the resultant P_{K_d} to the main process. According to equation (4), adjacent partial matrices may share a common column in P. Therefore, P_{K_d} is an $N \times N_d$ matrix where $N_d = l_{j_{\text{first}}} - l_{j_{\text{last}}} + 1$ and j_{first} and j_{last} are the column indices for $\tilde{P}$ of the first and the last elements in K_d, respectively. When the main process receives P_{K_d}, the partial matrix P_{K_d} is shifted to the $l_{j_{\text{first}}}$-th column and added to matrix P. Figure 5 shows the case when the system matrix P in Figure 3 is executed in a distributed-memory system of two nodes. The general case is depicted in the flowchart in Figure 6. In our implementation, the distributed memory behavior is simulated by the operating system through MPI on a 2-processor-12-core machine.

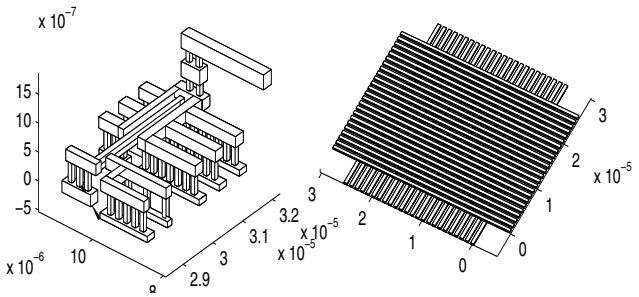

Figure 7: Transistor interconnect and 24×24 bus examples.

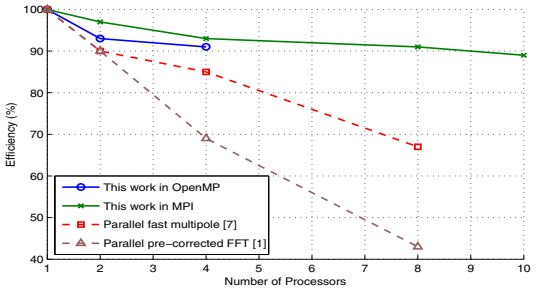

Figure 8: Parallel scalability for 24×24 buses.

Table 2: Performance improvement by using the integration acceleration techniques.

	FASTCAP	Instantiable basis func.		Impr.
	[4]	w/o accel.	w/ accel.	
Setup time		94.1 ms	50.7 ms	86%
Total time	340 ms	97.8 ms	54.4 ms	76%
Memory	24 MB	800 KB	2.5 MB	

Table 3: Performance for 24×24 buses in Figure 7.

# of used	Shared-memory system with 4 nodes			Dist.-memory system with 10 nodes		
nodes	Time	Speedup	Eff.	Time	Speedup	Eff.
1	40.5 s	1.00×	100%	44.1 s	1.00×	100%
2	21.7 s	1.86×	93%	22.7 s	1.94×	97%
4	11.1 s	3.65×	91%	12.3 s	3.56×	93%
8				6.04 s	7.30×	91%
10				4.95 s	8.91×	89%

6. EXAMPLES

We use an industry provided transistor interconnect structure in Figure 7 to demonstrate the effectiveness of the integration acceleration techniques. Our acceleration technique developed in Section 4 improves the system setup time by 86% and the overall computation in a single-threaded environment for this example is 6.2 times faster than FASTCAP [4] with a 10 times smaller memory requirement on a Xeon 3.2 GHz machine for the same 2.8% error. This accuracy is compared with a finely discretized FASTCAP reference solution which is obtained by refining the discretization by 10% for each iteration until the solutions from the last two iterations are within 0.1% difference. This result is summarized in Table 2.

We use another 24×24 bus structure in Figure 7 to show the parallel efficiency for different numbers of nodes in Figure 8. our solver achieves 91% parallel efficiency by using our shared-memory implementation on a Xeon 3.2 GHz 4-core machine and 89% by using our distributed-memory implementation on a Xeon 2.6 GHz 2-processor-10-core machine, whereas the parallel efficiencies of the parallel precorrected FFT [1] and of the parallel fast multipole expansion method [7] drop significantly to 42% and 65% at 8 cores, respectively. These efficiencies are the best available values for a much smaller 2×2 bus example from their original papers with medium discretization. The speedups and efficiencies of our solvers are listed in Table 3.

7. CONCLUSIONS

In this paper, we demonstrated that the parallelization bottleneck in traditional boundary element methods can be avoided by using instantiable basis functions. Our examples showed that our solver is 6× faster than FASTCAP in a single-thread environment. Furthermore, our approach achieved about 90% parallel efficiency with 10 parallel computing nodes through MPI implementation. We plan to release our solver on public domain as open source.

8. REFERENCES

[1] N. R. Aluru, V. B. Nadkarni, and J. White. A parallel precorrected fft based capacitance extraction program for signal integrity analysis. In *Proceedings of the 33rd annual Design Automation Conference*, DAC '96, pages 363–366, New York, NY, USA, 1996. ACM.

[2] B. Bond, Z. Mahmood, Y. Li, R. Sredojevic, A. Megretski, V. Stojanovi, Y. Avniel, and L. Daniel. Compact modeling of nonlinear analog circuits using system identification via semidefinite programming and incremental stability certification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(8):1149–1162, aug. 2010.

[3] Y.-C. Hsiao, T. El-Moselhy, and L. Daniel. Efficient capacitance solver for 3d interconnect based on template-instantiated basis functions. In *Electrical Performance of Electronic Packaging and Systems, 2009. EPEPS '09. IEEE 18th Conference on*, pages 179–182, Oct. 2009.

[4] K. Nabors and J. White. Fastcap: a multipole accelerated 3-d capacitance extraction program. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(11):1447–59, 1991.

[5] G. F. N. M. O. Vinyals. Revisiting a basic function on current cpus: A fast logarithm implementation with adjustable accuracy. *International Computer Science Institute*, 2007.

[6] J. Phillips and J. White. A precorrected-fft method for electrostatic analysis of complicated 3-d structures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(10):1059–72, 1997.

[7] Y. Yuan and P. Banerjee. A parallel implementation of a fast multipole-based 3-d capacitance extraction program on distributed memory multicomputers. *Journal of Parallel and Distributed Computing*, 61(12):1751–1774, 2001.